下载APP

28 | 程序可以在运行时进行链接吗？

2022-02-28 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 14:45 大小 13.52M

你好，我是于航。

在上一讲中，我介绍了有关 Linux 下静态链接的内容。而这一讲，我们将继续程序的“链接”之旅，来看看我之前提到的另外两种链接类型，加载时链接与运行时链接。

实际上，加载时链接与运行时链接均可归为动态链接，只是在这两种方式中，程序进行链接的具体时刻有所不同。其中，加载时链接发生在程序代码被真正执行之前；而运行时链接则可发生在程序运行过程中的任意时刻。

为什么要使用动态链接？

在上一讲中，我已经简单介绍了静态链接与动态链接两者的区别。其实，动态链接技术出现的最重要目的，便是为了解决静态链接具有的一些明显缺点。试想，假设一个应用程序依赖于多个第三方模块提供的函数实现，而这些模块均以静态库（包含有多个目标文件）的方式提供。那么，每次想要使用它们的最新版本时，我们都需要显式地将程序与它们重新进行链接。对于大多数普通的应用使用者来说，这个过程所花费的成本当然是无法接受的。

另外，使用完全静态链接也会导致那些本可以被多次重用的通用功能函数，无法被统一“提取出来”，这便会导致程序的二进制可执行文件体积变大。并且，这些通用代码的副本会随着多个进程的运行，被多次加载到内存中，而这也极大地浪费了宝贵的内存资源。而动态链接技术的出现，便可以解决上述这些问题。

能够使用动态链接加载的库被称为“共享库（Shared Library）”。在 Linux 中，这类库文件通常以 “.so” 后缀结尾。在深入介绍动态链接的基本原理之前，我们先来看看如何在真实项目中使用它。

使用共享库

回顾 25 讲 中的内容，我们能够知道，动态库本身也是 ELF 格式的一种具体文件类型，它对应着 elf.h 中的宏常量 ET_DYN。接下来，我仍以上一讲中的两段代码为例，来带你看看如何在实际项目中使用动态库。

这里，我们将把 sum.c 文件编译成动态库，并让 main.c 对应的应用程序使用。整个过程可以分为如下几步：

1. 使用命令 `gcc sum.c -shared -fPIC -o libsum.so` 将文件 sum.c 编译成名为 libsum.so 的动态库文件。这一步中使用的参数 “-shared” 表明创建一个动态库；参数 “-fPIC” 表明生成“位置无关代码”。关于这个选项的详细用途，我会稍后为你介绍。
2. 使用命令 `gcc -o main main.c -lsum -L.` 编译应用程序。这里我们将 main.c 与第一步生成的 libsum.so 共享库放在一起编译。命令中，参数 “-L.” 可用于为编译器指定更多的共享库查找目录，这里我们为其添加了 libsum.so 的所在目录；参数 “-l” 则用于指定需要参与编译的共享库，通过指定名称 “sum”，编译器会自动使用搜索到的，合法的 libsum.so 文件。

3. 使用命令 `export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH` 设置动态链接器在查找相关动态库时的位置。顾名思义，动态链接器是一段在程序运行时，用于帮助其查找所需共享库的代码。它在查找指定共享库文件时，会按照一定顺序，从多个不同位置进行查找。而这里通过 LD_LIBRARY_PATH 环境变量指定的位置，便是其中一个。
4. 使用命令 `./main` 运行程序。

需要注意的是，除了可以使用上述第三步介绍的“修改 LD_LIBRARY_PATH 变量”的方式来指定共享库的运行时查找目录外，我们还可以使用 rpath 和 ldconfig 这两种方式。它们分别通过“将动态库所在路径嵌入到可执行文件”，以及“将共享库安装到当前系统可访问的全局环境中”这两种方式，使得对应的共享库可以顺利地被动态链接器查找。这里，你可以直接点击对应的链接，来了解更多信息。

而为了让共享库真正地做到“可以被多个进程共享”，我们便需要让它的代码成为“位置无关代码”。下面我们来看看这个概念。

位置无关代码

位置无关代码（Position Independent Code，PIC）是一类特殊的机器代码，这些代码在使用时，可以被放置在每个进程 VAS 中的任意位置，而无需链接器对它内部引用的地址进行重定位。大多数现代 C 编译器在编译源代码时，均会默认产生这种 PIC 代码，而无需用户显式指定。当然，为了以防万一，你也可以通过添加 “-fPIC” 等参数的方式来明确指出。

通常来说，我们可以将模块（可以理解为独立的应用程序，或共享库）之间的数据引用分为四种方式：

模块内部的函数调用；

模块内部的数据访问；

模块之间的函数调用；

模块之间的数据访问。

其中，模块内部的函数调用在大多数情况下可以直接以 PC-relative 的寻址方式进行，因此它并不依赖于目标函数在整个进程 VAS 内的绝对地址。而对于模块内部的数据访问，由于

编译器在生成模块代码时，其 .data 与 .text 两个 Section 之间的相对位置是固定的，数据的访问也可以使用稳定的相对地址进行。总的来看，发生在模块内部的数据或函数资源引用，都不会因为模块代码被加载到进程 VAS 的不同地址而受到影响。但对于不同模块之间来说，事情就变得复杂了起来。

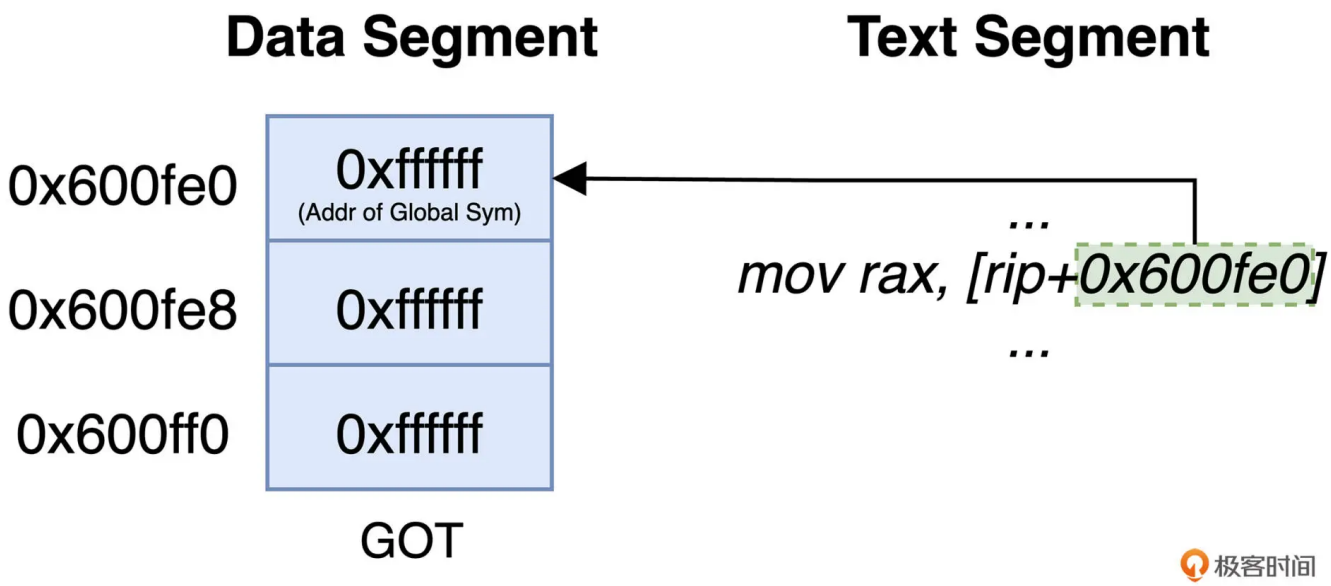
来看一个简单的例子。假设有一个共享库 M，它在内部的某个函数需要引用由应用程序定义的某个全局变量。而此时，程序 A 与 B 都想使用 M 中的这个函数。但相关的共享库代码（引用处）以及程序代码（被引用处），两者在进程 VAS 中的具体加载位置都并不确定。因此，在大多数情况下，两个程序对 M 中该变量引用地址的重定位修改值也并不相同。而这便会导致它们无法真正地共享同一份物理内存中模块 M 的代码。

PIC 的出现使得共享库代码可以做到真正地被多个进程复用，它利用了一个很简单的思想，即“将易变的部分抽离到进程独享的可修改内存中”。而为了做到这一点，编译器需要为各个模块添加额外的 Section 结构，这就是我接下来要讲的“全局偏移表”。

全局偏移表

全局偏移表（Global Offset Table，GOT）是位于每个模块 Data Segment 起始位置处的一个特殊表结构，其内部的每个表项中都存放有一个地址信息。而这些地址便分别对应于被当前模块引用的外部函数或变量在进程 VAS 中的实际地址。

模块在被编译时，其 Text Segment 与 GOT 之间的相对距离是能够计算出来的。因此，编译器可以利用这一点，来让代码直接引用 GOT 中的某个表项。同时，编译器还会为这些表项生成相应的重定位记录。这样，当程序被加载进内存时，动态链接器就可以根据实际情况，通过修正 GOT 表项中的值，来做到间接修正代码中对应符号的实际引用地址。你可以通过下图来直观地感受这个流程：



但需要注意的是，**并非所有编译器都会通过 GOT 来间接引用模块使用到的所有全局变量。**为了优化程序在某些特殊场景下的性能，编译器可能还会采用 Copy Relocation 等方式，来实现同样的效果。但对于外部函数的调用来说，GOT 在整个过程中仍然扮演着十分重要的角色。

过程链接表

虽然我们可以让动态链接器在程序加载时，将其代码中使用到的所有外部符号地址，更新在相应的 GOT 表项中，但当程序依赖的外部符号越来越多时，重定位的成本也会越来越高。而这便会导致程序初次运行时的“启动延迟”逐渐变大，甚至影响到程序正常功能的运作。为了解决这个问题，编译器为模块另外添加了名为“过程链接表（Procedure Linkage Table，PLT）”的 Section 结构。该表将协同 GOT，一起进行针对函数符号地址的“延迟绑定”。

PLT 是位于 Text Segment 中的一个表结构，其内部同样由众多表项组成。每个表项中都有着一段特殊的机器代码，用于完成相应任务。其中，PLT[0]（即 PLT 中的第一个表项，其他写法依此类推）较为特殊，它内部存放的代码专门用于调用动态链接器。而其他表项中则依次存放着，用于完成用户函数调用过程的相关代码。这些表项的地址将被程序中的 `call` 指令直接使用。

除此之外，在 ELF 文件中，GOT 对应的整个 Section 实际上被划分为更细致的 .got 与 .got.plt 两个部分。其中，前者主要用于保存相关全局变量的地址信息；而后者则主要参与

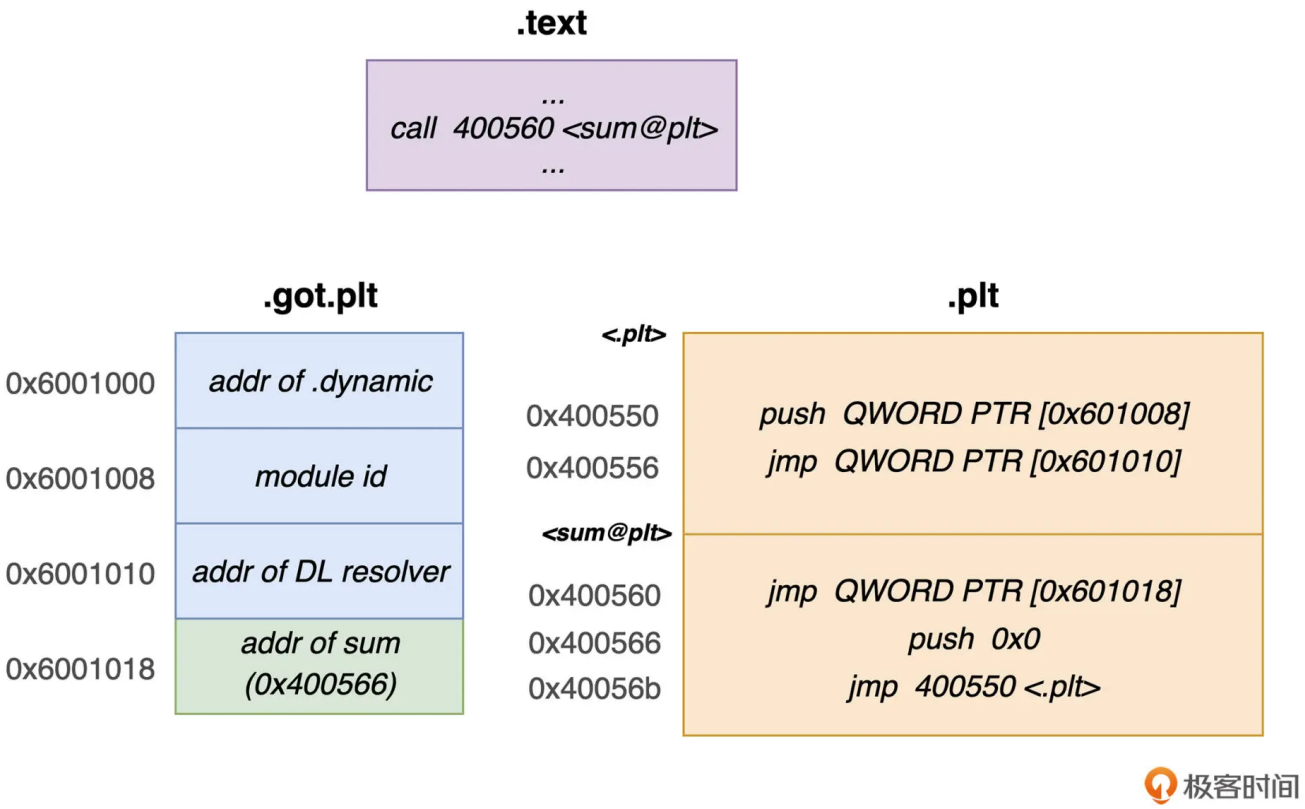
到函数符号的延迟绑定过程中。.got.plt 中的前三个表项具有特殊意义，它们保存的具体内容描述如下：

第一个表项中保存的是 .dynamic 的地址。这个 Section 中保存了动态链接器需要使用的一些信息；

第二个表项中保存的是当前模块的描述符 ID；

第三个表项中保存的是函数 _dl_runtime_resolve 的地址。该函数由操作系统的运行时环境提供，它将参与到 GOT 的运行时重定位过程中。

接下来，我们详细看看延迟绑定的具体执行过程。这里，我将以上面“使用共享库”小节中，共享库里 sum 函数的调用过程为例来进行介绍。你可以先看看下面的图片，对整体流程有个大致的感知，然后跟我具体来看每个步骤。



sum 函数的初次调用过程可以分为四步：

1. 程序通过 `call` 指令，调用对应于 sum 函数的 PLT 表项中的代码；
2. 该表项中的第一行代码（位于 0x400560）会通过 .got.plt 的第四个表项中的值进行间接跳转。该表项对应于函数 sum 的真实地址，但在第一次访问时，其值为对应 PLT 表

项中第二条指令的地址（即 0x400566）；

3. push 指令（位于 0x400566）将 sum 函数的 ID 压入栈中。通过 jmp 指令（位于 0x40056b），程序跳转到 PLT[0]；
4. push 指令（位于 0x400550）将 GOT[1] 中存放的模块描述符 ID 压入栈中，然后通过 jmp 指令（位于 0x400556）跳转到 GOT[2] 中存放的 _dl_runtime_resolve 函数的所在地址。该函数会使用当前存放于栈上的两个参数，来完成 sum 函数在 GOT 中的重定位。最后，它会将执行流程重新转移至 sum 函数内部。

至此，sum 函数的第一次执行便结束了。而在经过上述这一系列步骤后，sum 函数在整个进程 VAS 中的真实地址，便已经被更新到了对应的 GOT 表项中。因此，当它被再次访问时，程序仅通过以下这两个步骤便可完成调用：

1. 程序通过 call 指令调用 sum 函数对应 PLT 表项中的第一行代码（位于 0x400560）；
2. 该行 jmp 指令通过 sum 函数在 GOT 对应表项中已经修正的地址，间接跳转到该函数的第一行代码。

以上便是 sum 函数初次进行地址延迟绑定，以及再次访问时的整体流程。到这里，我已经为你介绍了动态链接的基本实现方式，下面我们来看看基于此进行的加载时链接与运行时链接这两者的主要区别。

加载时链接

实际上，加载时链接作为动态链接的一种具体类型，便是基于我上面介绍的 GOT 与 PLT 两个表结构进行的。它的一个最主要特征是，**动态链接器进行的符号重定位过程发生在程序代码被真正执行之前**。而为了做到这一点，操作系统执行应用程序的具体步骤也发生了改变。

操作系统内核在将应用程序装载到内存后，会根据其具体 ELF 类型的不同，来选择不同的处理方式。对于采用完全静态链接的可执行文件来说，内核会将控制权直接转移给应用程序，并执行其 Text Segment 中的入口代码。而对于使用了动态链接的可执行文件来说，在执行程序代码前，内核会首先根据名为 .interp 的 Section 中的内容，将相应的动态链接器共享库（ld.so）映射至进程的 VAS 中，并同时将控制权转移给它。

动态链接器在执行过程中，会通过其自身 .dynamic 中记录的信息，来完成对自己的重定位工作。接着，通过访问应用程序的 .dynamic，动态链接器可以获得它依赖的所有外部共享库，并在此基础之上完成对整个程序的动态链接过程。

运行时链接

顾名思义，运行时链接即符号的重定位发生在程序的运行过程中。这种方式有时也被称为“动态载入”或“运行时加载”，它的基本原理与正常的动态链接完全一致，只是链接的发生过程被推迟到了程序运行时。通过这种方式，程序可以自由选择想要加载的共享库模块，并在不使用时及时卸载，程序的模块化组织变得更加灵活。

运行时链接主要通过由动态链接器提供的四个 API，即 dlopen、dlsym、dlerror，以及 dlclose 来实现。来看一个简单的例子：

复制代码

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
typedef double (*cos_t)(double);
int main(void) {
  cos_t cosine;
  char *error;
  void* handle = dlopen("libm.so.6", RTLD_LAZY);
  if (!handle) {
    fprintf(stderr, "%s\n", dlerror());
    exit(EXIT_FAILURE);
  }
  dlerror();
  cosine = (cos_t) dlsym(handle, "cos");
  error = dlerror();
  if (error != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(EXIT_FAILURE);
  }
  printf("%f\n", (*cosine)(2.0));
  dlclose(handle);
  return 0;
}
```

这段代码的逻辑十分简单。我们通过“运行时链接”的方式，在程序的运行过程中从共享库文件 libm.so.6 内部加载了函数 cos。而在程序最后，我们使用实参 2.0 调用了这个函数，并打印出了执行结果。

这里，函数 dlopen 用于打开一个指定的共享库。通过它的第二个参数，我们能够指定符号重定位的具体执行方式。这里的 RTLD_LAZY 表示延迟绑定，即动态链接器仅会在特定函数被调用时，才对其使用到的相关符号进行解析。

函数 dlsym 则用于从一个打开的共享库实例中获取某个具体符号的地址。而在此之后，我们便能够以函数指针的形式对它进行调用。最后，当共享库使用完毕，通过 dlclose 函数，我们可以减少共享库实例的被引用次数。而当该次数变为 0，且共享库对象中的符号没有被其他对象引用时，该共享库对应内存便会从当前进程的 VAS 中被卸载，卸载的具体时机则由操作系统决定。

在上述整个流程中，我们可以使用 dlerror 函数，随时获取 dlopen API 函数在执行过程中产生的错误诊断信息。

总结

这一讲，我主要为你介绍了动态链接的基本实现方式，和基于此进行的加载时链接与运行时链接这两者的主要区别。

动态链接利用 GOT，将需要重定位的部分，分离到所在进程的 Data Segment，进而使得共享库文件可以被加载到进程 VAS 中的任意位置。在这种情况下，多个进程便能够做到真正地共享同一段物理内存中的共享库代码。而为了降低程序初次执行时，大量符号重定位带来的性能损耗，编译器又利用名为 PLT 的表结构，实现了对函数符号的延迟绑定。

加载时链接，是指在程序被真正执行前，动态链接器会首先完成对符号的重定位过程。而运行时链接则把这个过程推迟到了程序运行过程中，它的实现基于 dlopen、dlsym、dlerror，以及 dlclose 等几个动态链接器函数。

思考题

尝试了解一下 Linux 共享库使用的 soname 机制，并在留言区告诉我你的理解。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 编译器在链接程序时发生了什么？

下一篇 29 | C 程序的入口真的是 main 函数吗？

精选留言 (2)

写留言

liu_liu
2022-02-28

老师，有几个问题想了解下：

1. 假设共享库 M，引用由应用程序定义的某个全局变量 g。而 A、B 程序都链接了 M，那么 M 中的 .got 是不是需要复制一份？因为 A、B 中 g 变量的虚拟地址不一样。...

展开

作者回复: 这几个问题很棒！下面是回答：

1. 是这样的，每个进程的 VAS 中共享库 M 的内存映像内都有自己的 GOT Section，里面保存了不同的地址信息。在物理内存上看，Text Segment 中的数据是可以共享的，而 Data Segment 则是进程独立的。

2. 是的，通常动态链接器会默认使用第一个被 resolve 的符号，这个过程也被称为 “Symbol Interposition”。

3. 动态链接器通常会采用广度优先（breadth-first）的方式，按照可执行文件中 .dynamic Section 里 DT_NEEDED 条目内的记录顺序进行解析。

3

温长洋

2022-03-03

现在程序的链接库越来越多，请问老师，有没有办法找出程序所有存在symbol interposition的函数及所在的动态库。